从dedup说起

数组去重是一个基本的操作，方法众多：遍历去重到Set、Map去重、hashTable、Lodash Uniq，数组中是否存在对象、函数，每个去重方法的表现各有差异，本文将以此作为切入点深入源码进行分析。

1. **定义重复**

对于原始值而言，我们很容易想到1和1是相等的，'1'和'1'也是相等的。1和'1'是不相等的。那么对于如下情况呢？

1. NaN，NaN不是独立的数据类型，是数字类型，但是NaN不等于任何一个数字，也不等于任何一个其他的NaN，例如a=NaN，b=NaN，a== b不成立。但是这显然是不符合我们的判断是重复的需求的。好在js中有isNaN来弥补这个不足：isNaN(a) && isNaN(b)。
2. 对象和函数

初始化一个对象时，对象名放在桟中，它指向堆中存放的内容。所以两个对象直接对比是否相等是永远不等的，{a:1} == {a:1}永远不成立。函数也是类似的。要判断两个对象是否显式相等，我们可以采用遍历递归的方式来进行判断。

1. Null

Null并不是一个单独的特定类型，当对象的属性赋值为null时，表示该属性是空。Null的类型是object。

本文讨论ref去重，在这里不再赘述显式相等。有兴趣可以参看对象显式判等【代码1】

1. **数组去重**
2. **遍历（indexOf和includes）**

遍历是最容易想到也比较直观的方法：

代码如下：

function uniq1(arr) {

var newArr = [];

arr.forEach(function(item){

if(!newArr.includes(item)){

newArr.push(item);

}

});

return newArr ;

}

之所以使用includes而不使用indexOf方法，是因为对于NaN的判定，indexOf是不符合我们的去重规定的：

var arr = [1, 2, NaN];

arr.indexOf(NaN); // -1

arr.includes(NaN); // true

评价：includes是ES6的新语法，因此使用有局限性。另外不管是使用indexOf还是includes，本质上都是一个双重循环的方法，时间复杂度比较高，性能一般。

1. **hashTable方法**

简介一下hashTable方法：HashTable算法有一个规则，约束键与存储位置的关系。此方法就是利用键值对的对应关系，计算出一个hash值存为key，value存储将要判断的数据（数字或字符串）。因此通过使用HashTable结构记录已有的元素hash是一种典型以空间换时间的算法，其查找键值对的速度非常快。另外HashTable在JavaScript中实现较为简单，详见代码。

function uniq2(array) {

var newArr = []

var current\_temp = {}

for(var i = 0 ; i < array.length ; i++){

if(!current\_temp[array[i]]){

newArr.push(array[i])

current\_temp[array[i]] = true

}

}

return newArr

}

评价：需要注意的是，hashTable存放的key是不能区分对象和函数的，对象存储的key都是[object object]，函数存储的都是[function function]。

hashTable快于原生的Set方法，hashTable本身不存在对于Boolean、Number、String、NaN、undefined、null等类型的判断，一律会转为String，因此需要额外添加判断语句，本例hashTable方法还有许多不足。

1. **改进的hashTable方法**

如果像上文所说，hashTable方法是不完善的，我们可以通过手动添加条件，将hashTable方法完善起来。除了下面的方法可以完善hashTable，还有一种序列化Key的方法可以完善hashTable方法，有兴趣的可以查看【代码2】。

var u2 = (arr) => {  
 console.time('改进hashTable')  
 var temp\_num = {}  
 var temp\_string = {}  
 var has\_undefined = false  
 var has\_true = false  
 var has\_false = false  
 var newArrNormal = []  
 var newArrOthers = []  
 var current\_temp = temp\_num  
  
 for (var i = 0; i < arr.length; i++) {  
 if ((typeof arr[i] === 'string' || typeof arr[i] === 'number')) {  
 if (typeof arr[i] === 'string'){  
 current\_temp = temp\_string  
 }else if (typeof arr[i] === 'number'){  
 current\_temp = temp\_num  
 }  
  
 if(!current\_temp[arr[i]]){  
 newArrNormal.push(arr[i])  
 current\_temp[arr[i]] = true  
 }  
 }else if (arr[i] === undefined && !has\_undefined ){  
 newArrNormal.push(undefined)  
 has\_undefined = true  
 }else if (arr[i] === true && !has\_true){  
 newArrNormal.push(true)  
 has\_true = true  
 }else if (arr[i] === false && !has\_false ){  
 newArrNormal.push(false)  
 has\_false = true  
 }else if(!newArrOthers.includes(arr[i])){  
 newArrOthers.push(arr[i])  
 }  
 }  
  
 console.timeEnd('改进hashTable')  
 return [...newArrNormal, ...newArrOthers]  
}

总结：经过完善后的hashTable方法，增加了多次判断，因此性能应该大大降低。但是在只有number和String类型的情况下性能应该还是很快，此hashTable方法略有不足，会改变原数据的顺序。

**4.Map方法**

总结上述方法，对于key类型的限制，使得hashTable方法存在非常多的问题，那么有没有对key类型没有限制的对象呢？答案就是利用Es6的Map方法：Map是一种新的数据类型，可以把它想象成key类型没有限制的对象。此外，它的存取使用单独的get()、set()、has()接口。使用原生的Map方法时，作为Object的key ，Map是ref内存地址，但是object和func 会执行toString方法，导致速度变慢。这里需要特别说明的是，Map的get速度比Object快，但是set方法在Chrome内部是用js实现的，没有直接用Object方法。包了一层，所以速度会比较慢。

function uniq4(array) {

const seen = new Map()

array.filter((a) => !seen.has(a) && seen.set(a, 1))

return array;

}

1. **自己构建Map方法**

鉴于Chrome内核实现Map方法需要包一层js实现，我们可以自己使用Object方法直接进行对象数据的读取。Map实现代码见【附录2】

function myUniq(array) {

console.time('我的Map')

var seen = new myMap()

var newArr = []

array.filter((a) => !seen.has(a) && seen.set(a, 1))

console.timeEnd('我的Map')

return newArr

}

**5.Set方法**

利用Es6的Set方法，Set将数组转换为一个不含有重复元素的对象，然后再使用Array.from方法转换为数组。因为 Set 中的值总是唯一的，所以需要判断两个值是否相等。判断相等的算法与严格相等（===操作符）不同。具体来说，对于 Set ， +0 （+0 严格相等于-0）和-0是不同的值。尽管在最新的 ECMAScript 6规范中这点已被更改。从Gecko 29.0和 [recent nightly Chrome](https://code.google.com/p/v8/issues/detail?id=3069)开始，Set 视 +0 和 -0 为相同的值。另外，NaN和undefined都可以被存储在Set 中， NaN之间被视为相同的值（尽管 NaN !== NaN）。综上所述，Set方式可以说是最全面的数组去重的方法。

function uniq5(array) {

return Array.from(new Set(array));

}

1. **裸露的Map方法**

不同的浏览器对于方法是否包裹在函数内部有针对性的优化，例如谷歌浏览器。

代码如下：

//Map裸露

const seen = new Map()

array.filter((a) => !seen.has(a) && seen.set(a, 1))

1. **裸露的Set方法**

同上，代码如下：

//Set裸露

arr = [...new Set(arr2)];

1. **uniq方法**

使用lodash库的uniq方法，Lodash用来操作对象和集合,比Underscore拥有更多的功能和更好的性能。查看uniq源码详见【代码3】可以发现，uniq是分了情况来判断是否相等的，另外该方法对于NaN类型是支持的。该方法在数组内数据量小于200的时候使用的是直接递归判断（outer），当数据量大于200的时候，使用的是原生的Map方法。设置为200为界限值是作者经过大量测试的结果。

function uniq6(array) {

return \_.uniq(array);

}

1. **测试**
2. 测试用例

示例：

测试数组：[true, false, false, 1, "1", "1", 0, 0, "0", "0", undefined, undefined, null, null, NaN, NaN, () => (false),() => (false),Array(0), Array(0), /a/, /a/]

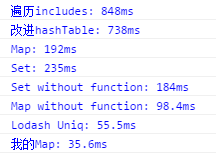
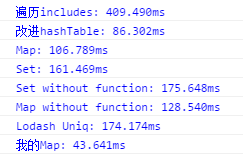
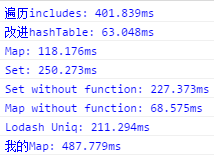
预期结果：[true, false, 1, "1", 0, "0", undefined, NaN, null, () => (false),() => (false), Array(0), Array(0), /a/, /a/]

实际测试数据量：

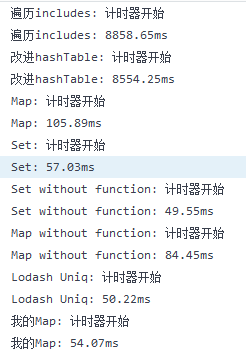
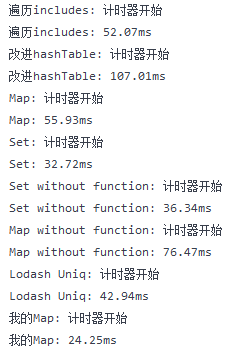
90万数字、2万字符串、2万对象、2万函数、1万undefined、1万null，合计100万

1. 测试结果

测试环境：Win7 64位 8G内存 Core i7 5320 数据量（100万） Chrome浏览器



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 遍历（includes） | hashTable（改进） | Map | Set | Map without function | Set without function | Loda  Uniq | MyMap |
| Number only | 401.839ms | 63.048ms | 118.176ms | 250.273ms | 68.575ms | 227.373ms | 211.294ms | 487.779 |
| Number & String | 409.490ms | 86.302 | 106.789ms | 161.469ms | 128.540ms | 175.648ms | 174.174ms | 43.641ms |
| Number & String & Object & Function | 848ms | 738ms | 192ms | 235ms | 98.4ms | 184ms | 55.5ms | 35.6ms |

测试环境：Win7 64位 8G内存 Core i7 5320 数据量（100万） Firefox浏览器 

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 遍历（includes） | hashTable（改进） | Map | Set | Map without function | Set without function | Loda  Uniq | MyMap |
| Number only | 32.67ms | 84.15ms | 55.36ms | 33.57ms | 89.81ms | 38.59ms | 49.96ms | 18.38ms |
| Number & String | 52.07ms | 107.01ms | 55.93ms | 32.72ms | 76.47ms | 36.34ms | 42.94ms | 24.25ms |
| Number & String & Object & Function | 8858ms | 8554ms | 105.89ms | 57.03ms | 84.45ms | 49.55ms | 50.22ms | 54.07ms |

1. 测试总结

测试代码详见【附录1】。

只有数字和字符串的类型的测试中，Chrome浏览器hashTable方法都是最快的，甚至远远快于原生的set方法。而在Firefox中，对ES6的原生方法includes支持是最快的，set方法和includes方法很接近。对于includes方法，Firefox的底层实现是直接取对象而不是通过指针读取数据。对于改进的HashTable，Firefox要慢很多，个人判断是对于if的执行语句Firefox是同步执行的，而chrome是异步执行。

在含有对象和函数的类型中，Chrome和Firefox都是lodash的uniq方法较为出色，该方法

性能也很稳定。

此外，因为不同的浏览器对裸露函数和包裹函数的执行进行过各自的优化，可以看出Chrome浏览器在这方面要优于Firefox，同时可以看到，包裹在Function里面的执行方法与裸露的方法优化是根据不同的方法进行的优化。

关于自己写的Map方法，是采用了直接Object读取的方式，因此速度较快。

【代码1】

**深度递归判断对象是否显式相等**

deepEquals = (obj1, obj2) => {  
 if (obj1 === obj2 || (isNaN(obj1) && isNaN(obj2))) return true;  
 if (typeof obj1 !== typeof obj2 || obj1 === null || obj2 === null) {  
 return false;  
 } else if (typeof obj1 === 'object') {  
 const keys1 = Object.keys(obj1);  
 const keys2 = Object.keys(obj2);  
 if (keys1.length !== keys2.length) return false;  
  
 for (let i = keys1.length - 1; i >= 0; i -= 1) {  
 const key = keys1[i];  
 if (!deepEquals(obj1[key], obj2[key])) return false;  
 }  
 return true;  
 }  
 return false;  
};

【代码2】

**在允许对对象进行自定义的比较规则，也可以将对象序列化之后作为key来使用。这里为简单起见，使用JSON.stringify()进行序列化。对于深度去重的JSON.Stringfy方法，因为需要序列号，时间复杂度与双重循环一样，性能较差，并且不能判定是否是function，需要额外使用toString判断function，而js不能获取ref的内存地址，除了 === 不能判断是否是同一个函数和对象，function toString 不能判断ref是否相同，因此建议非特殊局限不要使用此方法。**

function uniq3(arr) {

var ret = [];

var len = arr.length;

var tmp = {};

var tmpKey;

for(var i=0; i<len; i++){

tmpKey = typeof arr[i] + JSON.stringify(arr[i]);

if(!tmp[tmpKey]){

tmp[tmpKey] = 1;

ret.push(arr[i]);

}

}

return ret;

}

【代码3】

**lodash源码**

var createSet = !(Set && (1 / setToArray(new Set([,-0]))[1]) == INFINITY) ? noop : function(values) {  
 return new Set(values);  
 };

function uniq(array) {

return (array && array.length) ? baseUniq(array) : [];

}

function baseUniq(array, iteratee, comparator) {

var index = -1,

includes = arrayIncludes,

length = array.length,

isCommon = true,

result = [],

seen = result;

if (comparator) {

isCommon = false;

includes = arrayIncludesWith;

}

else if (length >= LARGE\_ARRAY\_SIZE) {

var set = iteratee ? null : createSet(array);

if (set) {

return setToArray(set);

}

isCommon = false;

includes = cacheHas;

seen = new SetCache;

}

else {

seen = iteratee ? [] : result;

}

outer:

while (++index < length) {

var value = array[index],

computed = iteratee ? iteratee(value) : value;

value = (comparator || value !== 0) ? value : 0;

if (isCommon && computed === computed) {

var seenIndex = seen.length;

while (seenIndex--) {

if (seen[seenIndex] === computed) {

continue outer;

}

}

if (iteratee) {

seen.push(computed);

}

result.push(value);

}

else if (!includes(seen, computed, comparator)) {

if (seen !== result) {

seen.push(computed);

}

result.push(value);

}

}

return result;

}

【附录1】

**测试代码：**

<!DOCTYPE html>  
<html lang="en">  
<script src="https://cdn.bootcss.com/lodash.js/4.17.4/lodash.js"></script>

<script src="./myMap.js"></script>  
<head>  
 <meta charset="UTF-8">  
 <title>Document</title>  
</head>  
<style>  
</style>  
  
<body>  
<script type="text/javascript">  
 var arr1 = []; //创建一个要去重的数组  
 var objArr = [];  
 var funArr = [];  
 var stringArr = [];  
  
 //数字  
 for (var i = 0; i < 900000; i++) {  
 arr1.push(parseInt(Math.random() \* 10) + 1);  
 }  
 //字符串  
 for (var i = 0; i < 20000; i++) {  
 stringArr.push((parseInt(Math.random() \* 10) + 1).toString())  
 }  
 for (var i = 0; i < 20000; i++) {  
 arr1.push(stringArr[i]);  
 }  
 //对象  
 for (var i = 0; i < 20000; i++) {  
 objArr.push({a: (parseInt(Math.random() \* 10) + 1)})  
 }  
 for (var i = 0; i < 20000; i++) {  
 arr1.push(objArr[i]);  
 }  
 //函数  
 for (var i = 0; i < 20000; i++) {  
 var a = () => (parseInt(Math.random() \* 10) + 1)  
 funArr.push(a)  
 }  
 for (var i = 0; i < 20000; i++) {  
 arr1.push(funArr[i]);  
 }  
 //undefined  
 for (var i = 0; i < 10000; i++) {  
 funArr.push(undefined)  
 }  
 for (var i = 0; i < 20000; i++) {  
 arr1.push(funArr[i]);  
 }  
 //null  
 for (var i = 0; i < 10000; i++) {  
 funArr.push(null)  
 }  
 for (var i = 0; i < 20000; i++) {  
 arr1.push(funArr[i]);  
 }  
  
  
 arr2 = arr1.concat()  
 arr3 = arr1.concat()  
 var u1 = (arr) => {  
 console.time('遍历includes')  
 var newArr = [];  
 arr.forEach(function(item){  
 if(!newArr.includes(item)){  
 newArr.push(item);  
 }  
 });  
 console.timeEnd('遍历includes')  
 return true ;  
 }  
  
 var u2 = (arr) => {  
 console.time('改进hashTable')  
 var temp\_num = {}  
 var temp\_string = {}  
 var has\_undefined = false  
 var has\_true = false  
 var has\_false = false  
 var newArrNormal = []  
 var newArrOthers = []  
 var current\_temp = temp\_num  
  
 for (var i = 0; i < arr.length; i++) {  
 if ((typeof arr[i] === 'string' || typeof arr[i] === 'number')) {  
 if (typeof arr[i] === 'string'){  
 current\_temp = temp\_string  
 }else if (typeof arr[i] === 'number'){  
 current\_temp = temp\_num  
 }  
  
 if(!current\_temp[arr[i]]){  
 newArrNormal.push(arr[i])  
 current\_temp[arr[i]] = true  
 }  
 }else if (arr[i] === undefined && !has\_undefined ){  
 newArrNormal.push(undefined)  
 has\_undefined = true  
 }else if (arr[i] === true && !has\_true){  
 newArrNormal.push(true)  
 has\_true = true  
 }else if (arr[i] === false && !has\_false ){  
 newArrNormal.push(false)  
 has\_false = true  
 }else if(!newArrOthers.includes(arr[i])){  
 newArrOthers.push(arr[i])  
 }  
 }  
  
 console.timeEnd('改进hashTable')  
 return [...newArrNormal, ...newArrOthers]  
 }  
  
 var u3 = (array) => {  
 console.time('Map')  
 const seen = new Map()  
 array.filter((a) => !seen.has(a) && seen.set(a, 1))  
 console.timeEnd('Map')  
 return true  
 }  
  
 var u4 = (arr) => {  
 console.time('Set')  
 arr = Array.from(new Set(arr))  
 console.timeEnd('Set')  
 return true  
 }  
  
  
 var u5 = (array) => {  
 console.time('Lodash Uniq')  
 \_.uniq(array)  
 console.timeEnd('Lodash Uniq')  
 return true  
 }  
  
 var myUniq = (array) => {

console.time('我的Map')

var seen = new myMap()

array.filter((a) => !seen.has(a) && seen.set(a, 1))

console.timeEnd('我的Map')

return true

}

u1(arr1)  
 u2(arr1)  
 u3(arr1)  
 u4(arr1)  
 console.time('Set without function')  
 arr = [...new Set(arr2)];  
 console.timeEnd('Set without function')  
  
 console.time('Map without function')  
 const seen = new Map()  
 arr3.filter((a) => !seen.has(a) && seen.set(a, 1))  
 console.timeEnd('Map without function')  
  
 u5(arr1)  
 myUniq(arr1)  
</script>  
</body>  
  
</html>

【附录2】

**自己实现的Map方法**

(function(window) {

var \_\_global\_id\_\_ = 1

var MapProto = function() {

var objectKeysMap = {}

var objectMap = {}

var normalMap = {

'string': {},

'number': {}

}

var Map = {}

Map.set = function(key, val) {

var type = typeof key

if (type === 'string' || type === 'number') {

normalMap[type][key] = val

} else {

if(!key.\_\_id\_\_){

var id = \_\_global\_id\_\_++;

key.\_\_id\_\_ = id

objectKeysMap[id] = key

objectMap[id] = key

}

}

}

Map.get = function(key) {

var type = typeof key

if (type === 'string' || type === 'number') {

return normalMap[type][key] === undefined ? null : normalMap[type][key]

} else {

var id = key.\_\_id\_\_

if (!id)

return null

else

return objectMap[id]

}

}

Map.has = function(key) {

var type = typeof key

if (type === 'string' || type === 'number') {

return normalMap[type][key] === undefined ? false : true

} else {

var id = key.\_\_id\_\_

if (!id)

return false

else

return true

}

}

return Map

}

var Map = function() {

if(this.constructor != Map)

throw new TypeError("Constructor Map requires 'new'")

}

Map.prototype = MapProto()

Map.prototype.constructor = Map

window.myMap = Map

})(window)